

AD-A049 472

TRW DEFENSE AND SPACE SYSTEMS GROUP REDONDO BEACH CALIF
IMPROVEMENTS TO SEMANOL. VOLUME I. (U)
NOV 77 P T BERNING

F/G 9/2

F30602-76-C-0238

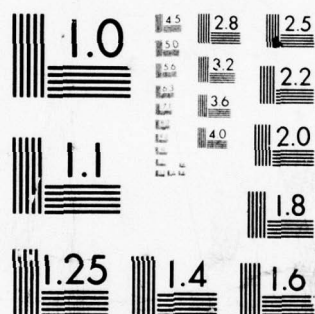
UNCLASSIFIED

RADC-TR-77-365-VOL-1

NL

1 OF 1
AD
A049 472





MICROCOPY RESOLUTION TEST CHART
NATIONAL BUREAU OF STANDARDS-1963-A

AD A049472

RADC-TR-77-365, Vol I (of four)
Final Technical Report
November 1977

2



IMPROVEMENTS TO SEMANOL

Paul T. Berning

TRW Defense and Space Systems Group

AD No.
 JDC FILE COPY

A049473

Approved for public release; distribution unlimited.

ROME AIR DEVELOPMENT CENTER
Air Force Systems Command
Griffiss Air Force Base, New York 13441

DDC
RECEIVED
FEB 3 1978
D

This report has been reviewed by the RADC Information Office (OI) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.

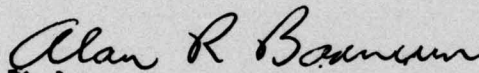
RADC-TR-77-365, Vol I (of four) has been reviewed and approved for publication.

APPROVED:



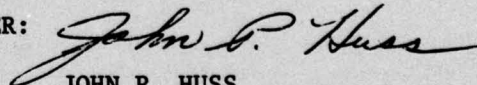
JOHN M. IVES, Captain, USAF
Project Engineer

APPROVED:



ALAN R. BARNUM, Assistant Chief
Information Sciences Division

FOR THE COMMANDER:



JOHN P. HUSS
Acting Chief, Plans Office

If your address has changed or if you wish to be removed from the RADC mailing list, or if the addressee is no longer employed by your organization, please notify RADC (ISIS) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return this copy. Retain or destroy.

UNCLASSIFIED

TR-77-365-VOL-1

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER RADC-TR-77-365, Vol I (of four)	2. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) IMPROVEMENTS TO SEMANOL - Volume I.	5. TYPE OF REPORT & PERIOD COVERED Final Technical Report May 1976 - May 1977	6. PERFORMING ORG. REPORT NUMBER N/A
7. AUTHOR(s) Paul T. Berning	8. CONTRACT OR GRANT NUMBER(s) F30602-76-C-0238	9. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS P.E. 63728F J.O. 5550840
9. PERFORMING ORGANIZATION NAME AND ADDRESS TRW Defense and Space Systems Group One Space Park Redondo Beach CA 90278	10. CONTROLLING OFFICE NAME AND ADDRESS Rome Air Development Center (ISIS) Griffiss AFB NY 13441	11. REPORT DATE Nov 1977 12. NUMBER OF PAGES 57
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Same	15. SECURITY CLASS. (of this report) UNCLASSIFIED 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.		
17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report) Same		
18. SUPPLEMENTARY NOTES RADC Project Engineer: Captain John M. Ives (ISIS)		
19. KEY WORDS (Continue on reverse side if necessary and identify by block number) SEMANOL syntax interpreter SEMANOL(76) language definition JOVIAL standardization JOVIAL(J3) language control semantics metalanguage		
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) This report describes the work that was performed in meeting the goals of this contract; provides a brief introduction to the SEMANOL method, and discusses issues in formal semantic description that arose in performing these tasks. This project accomplished the (1) definition of an improved semantic specification metalanguage, SEMANOL(76); (2) corollary upgrading of the Interpreter program to process the new metalanguage; (3) substantial improvement of the processing efficiency of this new SEMANOL(76) Interpreter; (4) implementation of a powerful new user command language for the SEMANOL(76) Interpreter;		

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

409637

HLL

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

(5) writing of a comprehensive formal specification of the JOVIAL(J3) programming language; and (6) presentation of a three-day course in the use of SEMANOL(76). The effectiveness of the SEMANOL system was greatly improved as a result of this work. In addition, a basis for control of the JOVIAL(J3) programming language was established by production of a formal specification of JOVIAL(J3). We believe SEMANOL(76) can now be a useful working tool in the USAF Higher Order Language Control Facility.

ADDITIONAL	
WTS	Write Section <input checked="" type="checkbox"/>
DBS	Self Section <input type="checkbox"/>
EXAMINATIONS	<input type="checkbox"/>
JUSTIFICATION	
BY	
DISTRIBUTION/AVAILABILITY CODES	
Dist.	AVAIL. AND/OR SPECIAL
A	

DDC
RECEIVED
FEB 3 1978
RECEIVED
D

UNCLASSIFIED

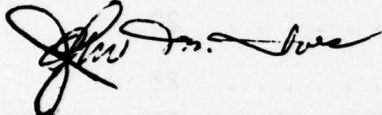
SECURITY CLASSIFICATION OF THIS PAGE(When Data Entered)

CONTENTS

	<u>PAGE</u>
<u>INTRODUCTION</u>	1
<u>THE NATURE OF THE SEMANOL SYSTEM</u>	6
<u>THE TASKS PERFORMED</u>	13
SEMAMOL(76) Developed	13
Interpreter Efficiency Improved	22
SIL Format Redesigned	22
Translator Improved	26
Executer Improved	30
Effective User Interface Implemented	31
Training Course Presented	34
JOVIAL(J3) Specification Written	39
<u>CONCLUSIONS AND RECOMMENDATIONS</u>	56

EVALUATION

This contract refined a previously operable programming language specification system into a more workable and more efficient system that will be more acceptable to the general software world. This specification system called SEMANOL (76), (an evolutionary update of SEMANOL (73)), since it is uniquely programmable and executable on a computer, allows side-by-side rigorous comparisons between the language specifications and the compiler written against it. While this is a very time-consuming task, even with the increased efficiency (approximating a 30-fold improvement), the addition of a trace facility output and a broad-resume capability allows the comparisons to be done on an interrupted basis, for more user convenience and completeness. Included in this effort was an enlarged rewrite of JOVIAL-J3 from the original SEMANOL to this newest version. In so doing it included implementation-dependent features such as COMPOOL and LOC that were not previously included and, thereby, laid a firm foundation for its formal use in the Higher Order Language JOVIAL facility, should it be so needed. SEMANOL (76) specification format will now be frozen as a stable product for continuity in future use.



JOHN M. IVES, Captain, USAF
Project Engineer

INTRODUCTION

SEMANOL is a TRW system of formally describing computer programming languages. Its formalism provides a descriptive precision that, at least in practice, cannot be obtained by the use of conventional, natural language, descriptions of semantics. Its formalism also permits any desired degree of descriptive completeness to be realized. Thus SEMANOL is meant to provide a means of alleviating many of the traditional problems related to programming language use and control. Such problems include:

1. The need to answer questions about a programming language through experimentation. Conventional reference documents for a language processor are sometimes incomplete or vague about the effect of executing a given language feature in a certain situation. The normal method of determining what the semantics are in such a case is to write a test program and observe what it does when executed. This is a tedious and difficult task, especially if done with the care it deserves, that ought not to be required. The results of such a test process must also be viewed with caution; since one is dealing with a poorly documented feature, its implementation may well stem from a poor specification and so contain a few "surprises".

2. The difficulty of transferring programs from one computer to another or even between different language processors for a single computer. The lack of uniformity in language is partly traceable to differences in underlying hardware operation and data representation, and so is essentially unavoidable. However, formal methods can permit the influence of such machine dependencies upon semantics to be made explicit, something ordinarily not done. The source of many other language differences is really a consequence of language implementors interpreting the design document in varying ways because of ambiguities in the natural language specification. This is a legacy of inadequate specification that formal methods directly seek to solve.

3. Compiler implementations that deviate from the customer's expectations. This is a problem in software procurement generally and, in the case of compilers, is due to programming language specifications being incomplete and ambiguous. Thus a variety of interpretations may arise that have some merit, with the particular problem that an implementor may make an interpretation different than the customer's interpretation without even being aware that something else might be meant. The restricted nature of programming languages suggests that improvement in this part of the software world should be possible.

4. The virtual impossibility of formulating standard definitions of programming languages that can truly serve as reference standards. A formal metalanguage is needed for precision and unambiguous interpretation, as prose text has been found unsuited for this task. The grave difficulties of implementation-defined aspects of a programming language also require a formalism in order that their intended existence and, perhaps, constraints upon their implementation may be made clear. A better system of definition can pay great dividends in standardization efforts.

These problems are familiar to most people who deal with programming languages and their processors, as are many other difficulties, and their ill effects upon software production are recognized (although often underestimated).

The SEMANOL system has been under development for several years and has enjoyed RADC support throughout. Its ability was established in earlier contracts, with the result that this project was undertaken with the intention of improving the SEMANOL system so that it could become a more useful tool in USAF programming language control. These improvements were designed to increase the operational convenience of the SEMANOL system, to provide faster computer execution of the system, and to expand the capabilities

of the metalanguage. The specific improvements made in this project include the following:

1. The definition of an improved semantic definition metalanguage. This metalanguage, dubbed SEMANOL(76), is an extension of SEMANOL(73), and reflects our increased understanding of formal semantic description. The changes made were based on our experience in using SEMANOL(73) to define JOVIAL(J73) and Universal CMS-2, and our analysis of JOVIAL(J3) requirements; thus they constitute an evolutionary refinement to the previously implemented metalanguage. The SEMANOL(76) metalanguage is believed to be easier to read and write than its predecessor, while also giving better processing efficiency through its inclusion of several new high level operators.

2. The implementation of an extensively revised Interpreter program that offers much better processing performance than did the earlier versions. Of course, the new Interpreter program also handles the SEMANOL(76) metalanguage. The processing efficiencies came from a certain amount of redesign, especially to make effective use of the Multics operating system upon which the Interpreter now operates; the use of a new intermediate language form; and the use of better coding techniques. A very substantial performance improvement was thus obtained.

3. The provision of a new user interface that supplies a convenient form of incremental translation and a powerful interactive metaprogram test system. The test system offers a flexible trace facility and a useful break capability; together, they provide users with a convenient, effective way in which to test formal specification metaprograms. User efficiency is thereby enhanced.

A more useful and acceptable system of semantic description has thus been created with the development of SEMANOL(76).

The improved SEMANOL(76) system was then used in the preparation of a comprehensive formal specification of JOVIAL(J3). This specification includes features such as COMPOOL, LOC, and overlay, and so includes programming language elements that are commonly left to be implementation defined. Storage allocation, arithmetic, and other such machine-determined details are likewise part of the specification that was written. The preparation of this specification raised various issues, especially several related to the best way with which to deal with flow control semantics and storage modeling in an involved language like JOVIAL(J3). That is, JOVIAL(J3) has its own unique features for which it is very difficult to divine sensible interpretations, particularly if one attempts to provide a generalized definition, as we did. This effort to achieve generality, and not just offer a personal definition, took a substantial amount of effort and was only partially successful; there remain open questions about how best to describe certain semantics.

The final accomplishment of this project was the presentation of a training course in the SEMANOL(76) method. Extensive materials were developed and used in the presentation of a three day training course given at RADC. The course introduced the SEMANOL(76) system and explained the ways in which it could be used. It provided technical details about the SEMANOL(76) metalanguage and the way in which we feel it should be employed when writing programming language specifications. A basis for expanding the guild of SEMANOL(76) users, and hence of SEMANOL(76) use, was thus established.

The new SEMANOL(76) metalanguage, the more efficient SEMANOL(76) Interpreter, the new Interpreter user command features, and the availability of training materials, all products generated in performance of this contract, together provide a greatly improved system of formal semantic description. Additionally, the SEMANOL(76) specification of JOVIAL(J3) provides a vehicle for practical evaluation of the SEMANOL(76) method as

well as supplying a basis upon which to construct an accepted, precise, programming language standard for JOVIAL(J3). SEMANOL(76) now is a useful, working, tool that is suitable for practical applications in programming language development and control.

THE NATURE OF THE SEMANOL SYSTEM

SEMANOL is intended for use in describing (procedural) programming languages. A specification written in the SEMANOL metalanguage is meant to provide an exact and complete definition of a programming language that is comprehensible to a suitably trained reader. That is, SEMANOL is designed to supply people with a basis for communication about programming languages that is more precise than commonly employed description methods. Additionally, the formality of the SEMANOL metalanguage permits operational use of the specification to be made upon a computer.

The SEMANOL specification method is algorithmic because it is felt that the semantics of programming languages ought to be explained in this way. That is, semantics are concerned with explaining how something happens and not just in characterizing an input-output relationship. Certainly this is the way in which language designers, compiler writers, and application programmers generally view the semantics of a programming language. Having a direct correspondence between the formal, operational, SEMANOL expression of language semantics and a reader's intuitive conception of a language yields a specification method that can be easily understood. An algorithmic method also permits language details, such as those specific to a given implementation, to be described exactly when desired.

The SEMANOL method considers a programming system, S , to be defined by $S = (P, I, T, \emptyset)$ where

- P = The set of programs which can be expressed in the programming system.
- I = The set of input values.
- T = The set of output traces. The trace is an ordered record of significant actions (such as assignment) that are performed by the program as it is executed; it is the visible manifestation of performing the algorithm that is the operational SEMANOL specification of semantics.

ϕ = The semantic operator. This operator, given as $\phi: P \times I \rightarrow T$, is considered to define the "meaning" of a program.

P, I, and T are each sets of strings which are specified by ϕ and whose individual members will be denoted by the corresponding lower case letters (i.e., $p \in P$, $i \in I$, $t \in T$). The effect of executing a given program, p, can then be denoted in terms of the semantic operator by

$$\phi(p, i) = t$$

Thus ϕ specifies the trace produced by any program in the system when that program is executed with any input value sequence. The SEMANOL meta-language is used for programming the semantic operator, thereby providing a method for formal specification of a programming language. Since SEMANOL is itself a programming language, it also belongs to a programming system. To differentiate between these two systems, we will use the subscript j to identify elements of the programming system being defined by a SEMANOL program and the subscript s to identify elements of the SEMANOL system. The semantics of a defined language program, p_j , are then expressed by

$$\phi_j(p_j, i_j) = t_j$$

The semantic operator for a defined language is expressed as a SEMANOL program, p_s , which in turn is interpreted by a semantic operator for the SEMANOL programming system, ϕ_s . Thus we have

$$\phi_s(p_s, (p_j, i_j)) = \phi_j(p_j, i_j) = t_j$$

and a formal definition of a defined language is provided by p_s . The SEMANOL semantic operator, ϕ_s , is defined in the SEMANOL Reference Manual and has been implemented by the SEMANOL Interpreter computer program.

This general view of language definition is shown graphically in Figure 1. As shown there, these levels of semantic specification correspond to defining a virtual machine for SEMANOL and, based on that, one for the language being defined.

A formal SEMANOL specification of a programming language is a metaprogram; a metaprogram for processing a source language program text written in the programming language being defined. The algorithm expressed by the SEMANOL metaprogram describes a way in which the intended effect of executing any program in the defined language can be realized. That is, the algorithm is an interpretive definition of semantics or, alternatively viewed, the metaprogram describes an interpreter for the defined programming language.

Now this metaprogram could certainly be written in more conventional programming languages, such as JOVIAL or Fortran; however, other programming languages, even those meant to do string processing, were not designed with formal semantic description in mind. Therefore, semantic interpreters would be difficult to write in these languages and, more importantly, the interpreters so expressed would be very difficult to understand. This lack of comprehensibility means these interpreters would serve poorly as specification standards. Contrarily, SEMANOL is a metalanguage specifically designed, and repeatedly refined, for expressing the semantics of programming languages. Because of this, an interpretive specification stated in SEMANOL is relatively easy to understand; the keywords and structure of SEMANOL, coupled with the use of proven specification conventions, provide a "naturalness" to the SEMANOL metaprogram that is not available with other interpreters. Precision is therewith combined with readability through the use of SEMANOL.

The SEMANOL metalanguage emphasizes high-level expressiveness. Where possible, "conventional" notation, as found in mathematical exposition and in other programming languages, is employed so that a reader's intuition will generally lead to a correct interpretation of SEMANOL code. The semantics of defined language execution are described by the use of SEMANOL in terms of parse trees and elements of the original source program text, and so can be directly understood by the reader.

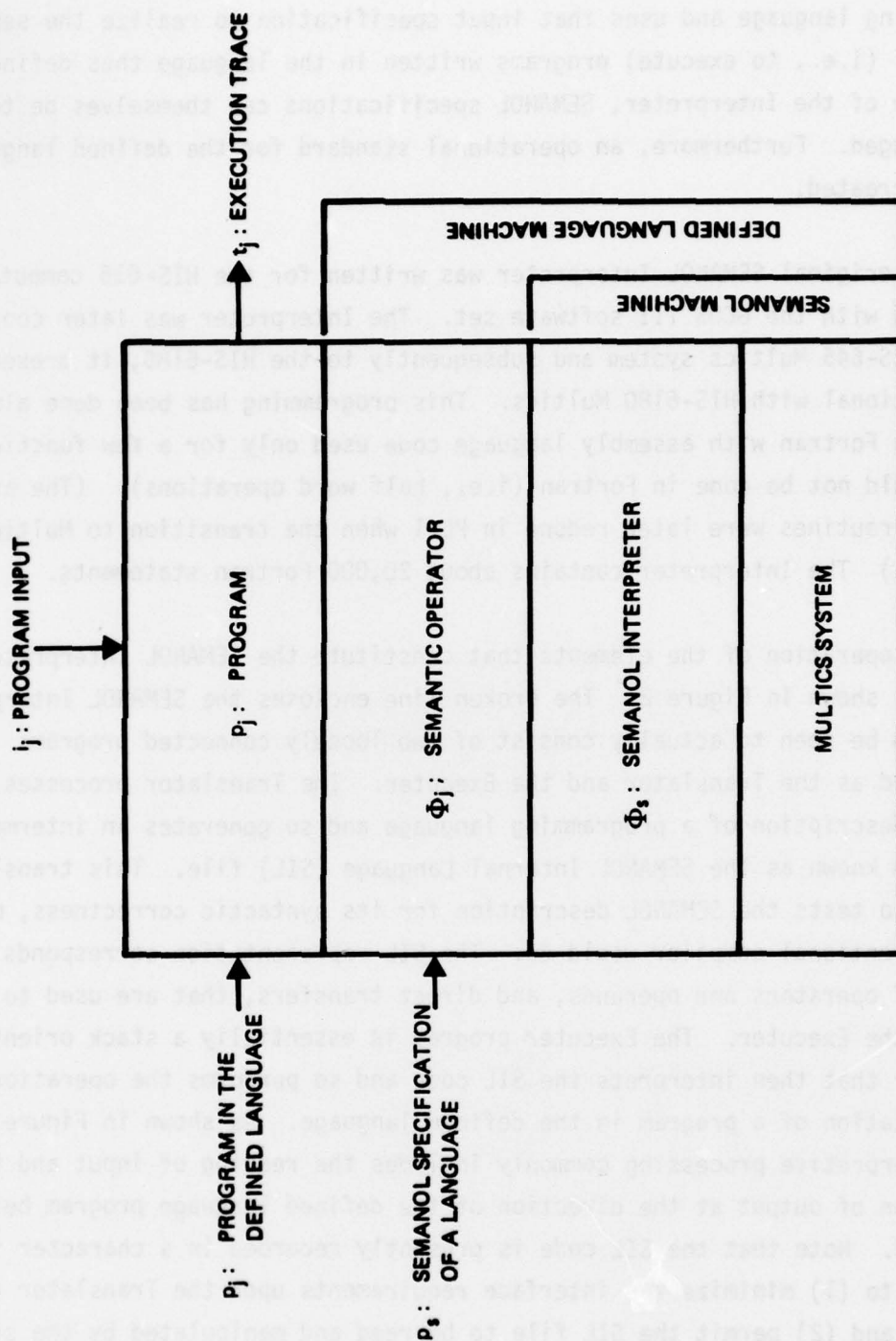


FIGURE 1. THE SEMANOL SYSTEM

The SEMANOL system has been implemented in the SEMANOL Interpreter program. The SEMANOL Interpreter accepts a SEMANOL specification of a programming language and uses that input specification to realize the semantic effect of (i.e., to execute) programs written in the language thus defined. By virtue of the Interpreter, SEMANOL specifications can themselves be tested and debugged. Furthermore, an operational standard for the defined language is thus created.

The original SEMANOL Interpreter was written for the HIS-635 computer operating with the GCOS III software set. The Interpreter was later converted to the HIS-645 Multics system and subsequently to the HIS-6180; it presently is operational with HIS-6180 Multics. This programming has been done almost wholly in Fortran with assembly language code used only for a few functions which could not be done in Fortran (i.e., half word operations). (The assembly language routines were later redone in PL/I when the transition to Multics was made.) The Interpreter contains about 20,000 Fortran statements.

The operation of the elements that constitute the SEMANOL Interpreter system is shown in Figure 2. The broken line encloses the SEMANOL Interpreter, which can be seen to actually consist of two loosely connected programs identified as the Translator and the Executer. The Translator processes the SEMANOL description of a programming language and so generates an intermediate code form known as the SEMANOL Internal Language (SIL) file. This translation phase also tests the SEMANOL description for its syntactic correctness, much as a conventional compiler would do. The SIL representation corresponds to a list of operators and operands, and direct transfers, that are used to control the Executer. The Executer program is essentially a stack oriented processor that then interprets the SIL code and so performs the operational interpretation of a program in the defined language. As shown in Figure 2, this interpretive processing commonly includes the reading of input and the production of output at the direction of the defined language program being processed. Note that the SIL code is presently recorded in a character format in order to (1) minimize the interface requirements upon the Translator and Executer and (2) permit the SIL file to be read and manipulated by the standard

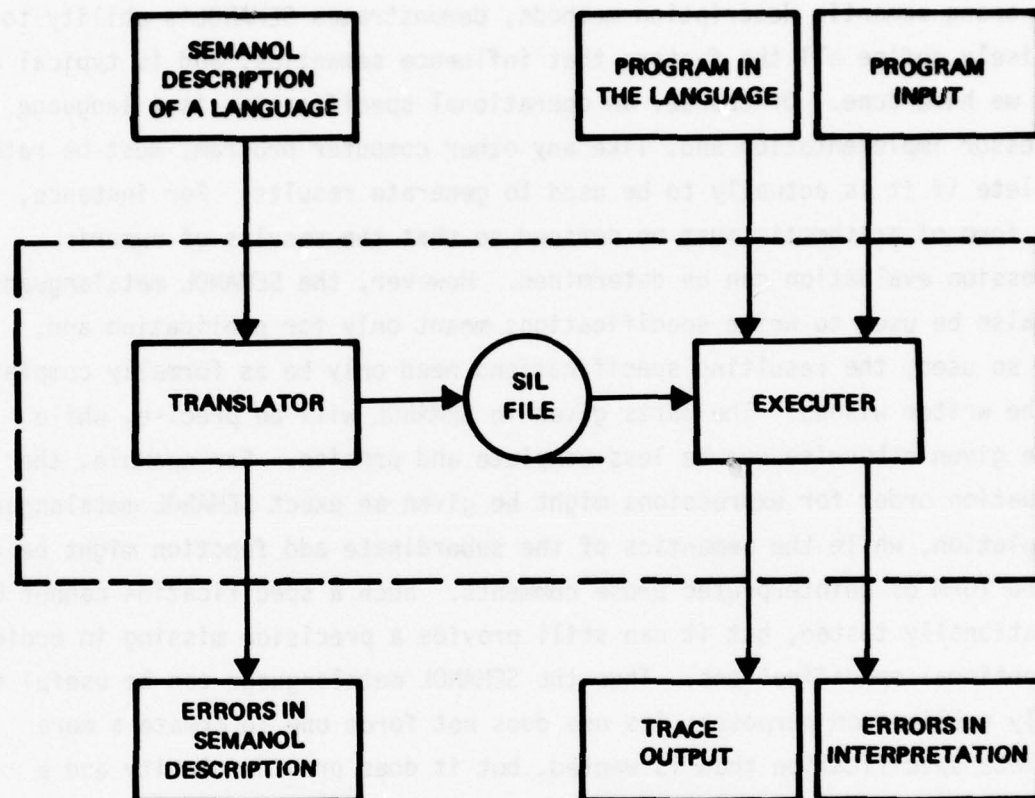


FIGURE 2. SEMANOL INTERPRETER ORGANIZATION

editing facilities of Multics. These attributes of the SIL file were very helpful throughout the SEMANOL Interpreter development period.

The discussion of this section has emphasized the use of SEMANOL in creating specifications of programming languages for which operational results can be achieved by use of the Interpreter computer program. We have emphasized this operational possibility of SEMANOL because it is useful, is rare among semantic description methods, demonstrates SEMANOL's ability to precisely define all the factors that influence semantics, and is typical of what we have done. Of course, an operational specification is a language processor implementation and, like any other computer program, must be rather complete if it is actually to be used to generate results. For instance, some form of arithmetic must be defined so that the results of numeric expression evaluation can be determined. However, the SEMANOL metalanguage can also be used to write specifications meant only for publication and, when so used, the resulting specifications need only be as formally complete as the writer wishes. The parts given in SEMANOL will be precise, while those given otherwise can be less complete and precise. For example, the evaluation order for expressions might be given an exact SEMANOL metalanguage formulation, while the semantics of the subordinate add function might be given in the form of uninterpreted prose comments. Such a specification cannot be operationally tested, but it can still provide a precision missing in ordinary conventional specifications. Thus the SEMANOL metalanguage can be useful for purely publication purposes; its use does not force one to create a more complete specification than is wanted, but it does provide clarity and a framework into which increasing detail can be added as appropriate.

THE TASKS PERFORMED

A variety of things were done to the existing SEMANOL(73) system in order to improve its usefulness and acceptability to users. The changes accomplished in this project expanded the capabilities of the system, made it more convenient to use, and substantially improved its execution efficiency. The system was also explained to possible users in a comprehensive training course that was given. In addition to general improvements to the system, the new SEMANOL(76) system was also used in preparing a formal specification of the JOVIAL(J3) programming language. Thus an improved system of semantic description was both developed and applied in performance of this contract. The manner in which this was done is described in what follows.

It should be noted that performance of these tasks required extensive use of the Multics system located at RADC. The Multics system was generally reliable and available when needed, and RADC assistance was always helpful and competent. The overall service was considered good.

SEMANOL(76) Developed

As a consequence of using SEMANOL(73) to describe JOVIAL(J73) and Universal CMS-2 in earlier contracts, and as a result of language analysis generally, it was felt that certain changes to the metalanguage would strengthen it. The changes were intended to make the metalanguage more naturally conform to the semantic description conventions that had evolved over past projects; specifications could thus more easily be read and written. Since the metalanguage was already a programming language of wide power, its descriptive domain was not extended by these changes. However, the convenience of semantic description was meant to be improved. These considerations thus led to the design of the SEMANOL(76) metalanguage.

In general, the changes and additions made to SEMANOL(73) in order to create SEMANOL(76) are individually rather minor, with many being essentially nothing more than syntactic variations of existing constructs. However, the cumulative effect was substantial and the metalanguage was changed more than we had originally intended. As always, the goal was to create a "final" version, so that the stability needed for SEMANOL(76) use in programming language standardization and control activities could be attained. Since a spark of invention remains, I do not imagine that we have succeeded in reaching that objective totally, but we have created a better metalanguage.

The most substantive metalanguage changes made to SEMANOL(73) were in the areas of procedure definition, invocation control, and arithmetic. In the case of arithmetic, the semantics of integer arithmetic were generalized to remove an existing host computer implementation dependency, while floating point arithmetic was simply deleted from SEMANOL(76). Specifically,

1. Integer arithmetic was re-implemented in SEMANOL(76) so that it operates upon integer operand strings of any length; that is, a true string arithmetic is provided. This change removed the SEMANOL(73) limits upon integer-range that were imposed by the past use of HIS-6180 double precision arithmetic. This extension now allows arithmetic of arbitrary operand lengths to be easily specified; SEMANOL(76) can thereby conveniently model a greater variety of real and abstract computers than could its predecessor.

2. Floating point constants and floating point arithmetic were then deleted from the metalanguage. In its SEMANOL(73) form, floating point arithmetic operand range and significance were limited. To convert floating point arithmetic to a completely machine independent form, as was done for the integers, would have been a lengthy task. It was considered unjustified since floating point arithmetic in defined languages can be conveniently modeled by integers (as is our accepted practice).

3. The new arithmetic of SEMANOL(76) caused the deletion of #R+, #R-, #R*, #R/, #FLOATING, #RNEG, #ENTIER, #NORMALIZE, #ROUND, and #HALF-ADJUST. The operators #ABS, #SIGN, and #CONVERT now apply only to integers.

These changes gave SEMANOL(76) an improved integer arithmetic, while deleting from the metalanguage the parameterized floating point arithmetic that was no longer being used.

The other major change made was the deletion of the abnormal return option from procedure definitions. SEMANOL(73) provided a #RESUME(;n) option that allowed return to be made from an invoked procedure to the ⁿth preceding procedure in the active call sequence. This feature had been used in past specifications (e.g., JOVIAL(J73)) solely when describing the control semantics of the programming language being defined. For this, it had been useful and had provided descriptive simplicity. Unfortunately, the semantics of #RESUME(;n) were themselves extremely complicated in SEMANOL(73). Thus a feature of a metalanguage designed to provide a clarity of semantic definition was itself likely to be misunderstood. While the issue of where one puts complexity is open to individual choice, the design of SEMANOL is based on the use of keywords and primitive ideas that are themselves simple and easily understood. By that criterion, a complex #RESUME(;n) feature was undesirable. It was thus removed from SEMANOL(76). The normal return option previously given by #RESUME(;0) was then replaced with #RETURN-WITH-VALUE. This new keyword also caused procedure definitions to return a value directly, and so corrected what had become recognized as a deficiency of #RESUME(;0). Procedure definitions had previously only been able to return values as side effects, and this had led to the generation of code that was less clear than it should have been.

A variety of lesser changes were also made as follows;

1. #PARENT-NODE and #ROOT-NODE were added to SEMANOL(76) so that parse trees can be traversed in an upward direction (they could already be traversed in a downward direction by use of #SEG). #PARENT-NODE returns the immediate antecedent node, while #ROOT-NODE returns the topmost (i.e., starting) node in the tree specified. These additions generalize the tree walking capability of SEMANOL(76) and can be especially useful in providing a simplification of syntactic components. Improved efficiency can also be realized through use of these new features.

2. A #SEQUENCE-OF-ANCESTORS-OF constructor was added to form a sequence of parent nodes for a given node. That is, the sequence is formed by traversal of the parse tree from the root node to a specified point.

3. #REVERSE-SEQUENCE was added to SEMANOL(76) and it has the obvious meaning; to create a sequence whose elements are an inversion of the specified sequence. The availability of this keyword can improve the efficiency of SEMANOL(76) programs.

4. A #WHILE statement was added to the control statements of SEMANOL(76); its effect is conventional. This addition improves the expressiveness of SEMANOL(76) with regards to loops (since only the #FOR-ALL and #IF were offered previously).

5. An inclusion relation was added that determines if an element is a node in a specified parse tree or not. This is done by means of the #IS #NODE-IN, or #IS-NOT #NODE-IN, relations. This feature enables testing for a node directly; formerly, a sequence of nodes would have had to be formed and the sequence then searched. Simplicity and efficiency are the products of this change.

6. A #FOR-ALL relation was added; this new feature complements the #THERE-EXISTS relation and so provides the ability to more simply express a much wider range of quantified relationships. This was a very natural extension to the SEMANOL(73) metalanguage.

7. #INPUT was changed so that it now reads the totality of program input (e.g., to the end-of-file) rather than one line at a time. This change means that the notion of "line", which is a characteristic of the programming language being described, is given explicitly in the SEMANOL(76) specification rather than being given a single definition by Interpreter convention. The information accepted by #INPUT must be in a standard Multics ASCII character format representation and so is capable of being created easily.

8. #OUTPUT was changed so that it simply concatenates a given string onto the current output string segment; it previously wrote fixed length strings on the standard output file. This change requires that the line terminators, and other print control codes, be inserted into the output string by explicit SEMANOL(76) metaprogram action. The ability to generate variable length output strings can simplify output formatting greatly, while also making the notion of a "line", as with #INPUT, a part of the SEMANOL(76) specification of a programming language. The output string is in a standard Multics ASCII character representation so that actual listing of the output string is easily done by an independent utility program; Multics already has the needed facilities for doing this. (Automatic listing of this output after Interpreter processing can be accomplished by the use of appropriate stored procedures.)

9. #GIVEN-PROGRAM was extended slightly so that it can be invoked more than once. However, it still reads only one program string, with the result that all invocations return the same value; i.e., the program string or #NIL, if there is no program string. The input string to #GIVEN-PROGRAM must be in standard Multics ASCII character format (as with #INPUT) and so may include non-printing characters such as carriage returns (CR) and line feeds (LF).

10. The method of describing strings was altered by the introduction of a uniform "escape" mechanism to deal with the representation of non-printing ASCII characters and single quotes within strings (strings continue to be delimited by single quote marks). Embedded quote marks were previously denoted by #L', #R', #LL', and #RR' conventions; this was dropped. Instead, square brackets are used to identify characters which cannot stand for themselves in strings; for example, 'ABC['D' rather than the older #L'ABC'D#R'. This new convention also allows non-printing ASCII characters, such as carriage returns, to be represented by multi-character names, but treated as the single characters they stand for. Now, a line of output may appear in a SEMANOL(76) program as 'THIS IS A TEST [CR]' while printing as THIS IS A TEST on a terminal line. All non-printing ASCII characters are provided in SEMANOL(76) with their standard abbreviated names. Note that these multi-character representations are mapped onto single characters (in the range 0-127) when carried in input-output files.

11. #EXTERNAL-CALL-OF was redefined to accept a sequence of strings as an input argument and to produce a string as the output value. The interface to external functions was implemented, for the first time, in performance of this contract.

12. #TCOPY was deleted. It no longer was used by the current technique of describing the semantics of control in a defined language.

13. The substring extractor operator #WORD-BETWEEN-FIRST...#AND-NEXT was deleted; it had not been used.

14. The tracing options, #TRACE-ON and #TRACE-OFF, are no longer part of the metalanguage; extended tracing options are provided with SEMANOL(76) but their activation is now accomplished by user commands (as discussed later).

15. #SPACE had been used to denote (1) the string consisting of one space character and (2) the set having as its only element a string of one space character. This second meaning of #SPACE was assigned to a new keyword #SPACESET, with #SPACE retaining just its first meaning. The SEMANOL(76) metalanguage was thus clarified.

16. #EMPTYSET was added to the syntax set-constants.

The new SEMANOL(76) metalanguage was used in this project to describe the JOVIAL(J3) programming language (and in a related contract, F30602-76-C-0245, to describe Minimal BASIC). This experience confirmed the desirability of making the SEMANOL(76) changes, although the influence that removal of the #RESUME(;n) feature would have upon describing JOVIAL(J3) control semantics was greatly underestimated. The previous semantic control models developed for JOVIAL(J73) and UCMS-2 could no longer be used, and the design of a new control model for the complicated JOVIAL(J3) programming language turned out to be very difficult. It is clear that this particular change made the writing of the JOVIAL(J3) specification far more difficult, but it is hoped that the result is more understandable to the reader than it would have been had #RESUME(;n) been used. Since realization of that hope is uncertain, it is possible that re-definition of #RESUME(;n), rather than outright deletion, might have been a better course. Further application and analysis of SEMANOL(76) will be needed to resolve that issue.

A variety of additional changes were made whose effects are primarily syntactic or, at least, rather modest. A reasonably comprehensive list of them follows;

1. The #AND operator was added. This operator is exactly the same in function as the '&', which is retained, but its inclusion does provide a more uniform set of Boolean operator names.
2. The syntax of enumerated sets was simplified slightly by requiring the use of the "<" wherever "[" could have appeared and ">" wherever "]" was allowed. These two notational forms were previously equivalent except in expressions involving #S-, where the [] form was required, so uniformity of notation is now realized. This change also permits the square brackets to be used in the revised form of string denotation described earlier in this section.
3. The syntax of string concatenation was restricted slightly by disallowing the #CW{a₁}...(a_n) form; #CW is now solely a binary operator.
4. The syntax of procedural definitions (i.e., #PROC-DF) was relaxed slightly in that the procedure body need no longer be included within #BEGIN, #END delimiters.
5. An extension of the permissible ordering of the sections of a SEMANOL program was made to allow the option of placing the <Control-section> before the <Semantic-definition-section>.
6. The previous #B+ was replaced with #BOR, #B- with #BXOR, and #B* with #BAND. The new forms are thought to be more descriptive of the operations performed than were the older keywords. The semantics were also changed in that leading zeros are no longer trimmed from the results produced by these three operators.

7. The syntax of control statements was changed slightly in that the '#' terminator is no longer used; the '#' was retained elsewhere, however.

8. The syntax of #INPUT and #OUTPUT was abbreviated slightly by deleting the #FROM and #TO phrases respectively. These phrases had no effect since standard files are always used.

9. The syntax of the #FOR-ALL clause was altered so that (1) its application to sequences could be more directly expressed and (2) it uses the newer notation for unbounded intervals.

10. The #C options for expressing set concatenation were deleted; these forms had become extraneous.

11. The #COMMENT keyword was dropped since the use of the alternative double quotes has been exclusively adopted (e.g., "This is a comment").

12. The syntactic classification of #UNDEFINED was changed, with the result that #UNDEFINED need not be parenthesized.

13. The syntax of sequence extraction was enlarged so that bounded intervals could be specified. This change makes SEMANOL(76) somewhat more uniform and also removes the only cases in which the #MIN and #MAX operators had been used. As a consequence of this, the #MIN and #MAX operators were dropped from SEMANOL(76).

14. #FIRST-CHARACTER-IN and #LAST-CHARACTER-IN were added to the substring extractor operators. These additions add no new capability, but they are slightly more readable than the previous forms. They are also more efficient than the previous equivalent forms.

Interpreter Efficiency Improved

The two major computer programs that form the SEMANOL Interpreter programming system, the Translator and the Executer, were conceived and originally implemented as part of a research project in formal programming language description (F30602-72-C-0047). Consequently, they were designed for ease of development and simplicity of modification; efficiency was not deliberately ignored, but it certainly was not a primary design factor. In addition, the original programs were written for a HIS-635 computer using the GCOS software product set then provided at RADC. This computer system was subsequently replaced with a succession of Multics systems and Fortran compilers, and the SEMANOL Interpreter was correspondingly converted to operate upon these new systems. However, these conversions were performed as simply as possible and with little attention to processing efficiency. Thus the SEMANOL Interpreter had evolved into a highly reliable set of computer programs that was recognized to be somewhat inefficient. One goal of this project was then to improve the efficiency of this system.

The most significant changes made in the pursuit of better efficiency are described in what follows. It should be observed that changes made for efficiency were often done concurrently with changes being made to support the new SEMANOL(76) metalanguage; the two issues are not entirely separable.

SIL Format Redesigned - SIL code is generated by the Translator program from SEMANOL program text, and is subsequently used to drive the Executer program; that is, the SIL code is interpreted by the Executer and so controls its operation. Thus SIL code is an intermediate representation of a SEMANOL program.

The SIL formats were necessarily altered to reflect the changes that were made in the SEMANOL metalanguage to create SEMANOL(76). However, two additional types of changes were made so that greater processing efficiency could be achieved and some simplification of the SIL realized. These changes were made to the SIL format generated for semantic procedural definitions and caused:

1. Replacement of the parenthesized, LISP-like, control level structure with a direct branching notation. The Executer's time to interpret control transfers is substantially reduced by this modification. This change also has the advantage that such direct transfers closely correspond to the operations available in the procedural languages (e.g., PL/1) into which we eventually hope to translate SEMANOL programs. Thus the Translator was brought closer to being able to generate executable code than it had been when producing the older notation.
2. Simplification of the method by which semantic and procedural definitions are activated by the Executer. The original method was modeled upon one used in SNOBOL; it caused argument and local variable names to be given global scope and so required the saving of the old values upon entry to a definition and their restoration upon exit. This generality was later found to be needless for SEMANOL and existing argument values are no longer saved and restored. Instead, arguments and local variables are referenced directly on the Executer stack. The generated code was also reduced so that the standard actions performed upon entering a semantic or procedural definition are now done automatically, using given parameters, rather than being done in response to explicit SIL code generated for that purpose. A reduction in interpretation overhead is thereby gained. The combination of these changes has made the Executer processing of semantic calls, a common activity when interpreting SEMANOL programs, much more efficient.

It should be observed that these changes also resulted in reducing the amount of SIL code produced; this reduction itself means that file space is used more effectively, and that a reduction of input-output transmission time is correspondingly achieved.

An example may help to illustrate just what has been done and why efficiency improvements have consequently been realized. For the SEMANOL semantic definition given by

```
#DF LONGER(StringA,StringB)
=> StringA #IF #LENGTH(StringA) >= #LENGTH(StringB);
=> StringB #OTHERWISE #.
```

the original Translator would have generated the following SIL code:

LONGER/SIL=

```
(1) (2 IPARAM/OP LONGER/VAR LOCAL/OP
(2) StringA/VAR PARAM/OP StringB/VAR PARAM/OP FCALL/OP
(3) ((StringA/VAR CVSTS/OP ISLEN/OP CVI/OP StringB/VAR
(4) CVSTS/OP ISLEN/OP CVI/OP PLT/OP LNOT/OP
(5) KTRUE/OP StringA/VAR LONGER/VAR ASVAR/OP ENDIF/OP)
(6) (StringB/VAR LONGER/VAR ASVAR/OP ENDIF/OP))
(7) RET/OP) ;
```

In the new system, the Translator generates SIL code as follows:

LONGER/SIL=

```
(1') [ 2 2 2
(3') 1/VAR CVSTS/OP ISLEN/OP CVI/OP 2/VAR
(4') CVSTS/OP ISLEN/OP CVI/OP PLT/OP LNOT/OP
(5') BFALSE/OP 3 1/VAR RET/OP
(6') 2/VAR RET/OP ] ;
```

(Note that SIL code is recorded as character strings since this form of representation minimizes interface requirements between the Translator and Executer programs, as well as otherwise aiding the development and checkout process. The parenthesized line numbers shown here are added for reference purposes and do not appear in the SIL code.)

The prologue of entry in the first case, lines (1) and (2), has been replaced by (1'). In both cases, the number of actual input parameters will be tested against the number expected, 2, but no longer is the IPARAM/OP given as an explicit operation to be invoked. Some overhead is thus now bypassed. Line (2) caused the old parameter values to be saved and the new ones to be assigned to the global variable names; such action is no longer done. The body of the definition, lines (3) and (4), is essentially unchanged although the references to arguments are now made directly, in line (3'), by position in the argument list (as held in the stack) rather than by global formal parameter name. Lines (5) and (6) display a branch option that employs the older, nested, control structure; a false result at KTRUE/OP will cause interpretation control to skip to the next list (i.e., line (6)). In the new method, a false result at BFALSE/OP causes a skip to the third entry that follows the 3 (i.e., to line (6')). The resulting economy in interpretation time is evident. It should be observed that while both SIL sequences end with RET/OP's, the function of this operator was changed substantially. RET/OP previously restored saved parameter values; it no longer does this. It was also changed to return the value at the top of the stack rather than the one assigned to the local variable having the name of the definition (LONGER in this example); an assignment operation, ASVAR/OP, can thereby be avoided. This is an efficiency gain that is not totally apparent in the SIL code itself. The other operators used in this example were not changed.

While the example demonstrates the advantage of direct branching with conditional tests, it should be realized that similar savings are gained when dealing with the code generated for SEMANOL(76) iterators.

Implementation of this new SIL format caused the code generation algorithms of the Translator to be extensively rewritten, often in conjunction with revisions being made otherwise for the new SEMANOL(76) features, and the main control loop of the Executer to be redone. Although the efficiency improvement in the Executer resulting from these changes was not rigorously measured, it was estimated that a two-fold reduction in processor time resulted from using this new SIL format. Note that most Executer operator routines were unaffected by the transition to this new intermediate form.

Translator Improved - A major effort was devoted to rewriting the recursive descent routines of the Translator. There is a recursive descent routine for each production, or group of similar productions, of the grammar of SEMANOL; their role is to analyze the syntax of SEMANOL statements and to determine the SIL code to be generated for each statement. These routines were, therefore, revised because of (1) the changes in the SEMANOL metalanguage that were made to create SEMANOL(76) and (2) the new SIL format that was introduced for better efficiency. These changes were pervasive and all recursive descent routines (there are about 50 of them) were revised. This revision was done with concern for the efficiency of the recursive descent routines themselves. It was also accompanied by a major improvement to the error messages produced and recovery procedures followed. Thus the user's efficiency was also enhanced.

While code analysis had suggested several areas in which improvements were possible within the Translator, timing tests had not been conducted. However, with Version 3.0 of Multics, it became easy to measure the relative usage of the subroutines of a program. Thus testing was then done with a SEMANOL(73) specification similar to the one of JOVIAL(J3) that was then being written for this project. The results of this process revealed that the lexical analyzer routines and the SIL output routines together used over 85% of the total execution time. Furthermore, one routine alone, SIL, used 40% of this time and another, SILPUT, used 26%. These two routines make use of Fortran input-output, and it became clear that Fortran input-output was much slower than had been imagined. While this general area was suspected to be a cause of inefficiency, the magnitude of the actual inefficiency was very surprising. Early attention was thus devoted to these two parts of the Translator.

The lexical analyzer routines of the Translator, of which there are five, scan SEMANOL statements and break them down into their constituent elements (called tokens). This process involves character string analysis and manipulation that in the original GCOS version of the Translator were done conveniently with the DECODE/ENCODE operators of Fortran. With the conversion to Multics Fortran, and the resulting absence of DECODE/ENCODE, the Translator

was expeditiously altered to use formatted READ/WRITE operations in conjunction with an intermediate file. This was the most convenient way in which to make the conversion, but it was inefficient. These routines were thus revised in this project so as to replace their use of Fortran input-output statements with references to equivalent PL/1 routines. The PL/1 routines are able to reference data directly, as based arrays, and to make use of the SUBSTRING operator. This procedure is much more direct than one using Fortran formatted input-output operations, and the resulting reduction in processor time for lexical analysis was dramatic.

The (eight) SIL output generator routines are responsible for collecting information and writing it on the SIL output file. In doing this, they hold information destined for the SIL file internally and often reorganize it before it is actually written on the SIL output file. This SIL information was stored internally by the Translator in text strings. In the original GCOS version of the Translator, these internal SIL strings were manipulated in storage by DECODE/ENCODE operators to produce the desired output format. Since Multics Fortran lacked DECODE/ENCODE operators, the Translator was later altered to use an intermediate external file and formatted READ/WRITE operations to achieve the same result. This was the most convenient way in which to make the program conversion, but the use of an intermediate file made this process slow. Therefore, efficiency was achieved by totally rewriting the SIL output routines so that no intermediate file was used. The use of an internal binary list representation, rather than character strings, for the SIL code was implemented while the external characteristics of the routines were retained. The binary data form can be manipulated in storage without recourse to reading and writing Fortran files, and so its handling is much faster. However, the choice of a new data representation was so fundamental to the operation of these routines that the code had to be entirely rewritten. Besides gaining greatly in efficiency, this new method yields a simpler design and more understandable code.

While these two sets of changes greatly improved Translator efficiency, several other modifications likewise had substantial efficiency benefits. The first of these was the retention of source names in generated SIL code. The Translator initially generated standardized names that were used in place of the names given in the original SEMANOL source text. The generated names then appeared in the SIL file passed to the Executer. The use of generated names shortened the SIL file and simplified the Translator logic slightly; however, the lack of SEMANOL source names during Executer processing complicated the debugging of SEMANOL specifications. A program called Tnames was then written as an independent program that restored the original names to lines in the SIL file and so improved the debugging process. By incorporating the function of Tnames in the Translator and deleting the distinct Tnames program, overall system efficiency, structure, and operational convenience were improved.

The Translator was also modified to use the recursive abilities of the host Multics system. The Translator makes extensive use of recursive subroutine calls despite being written in Fortran. Since GCOS Fortran did not support recursion (recursion normally not being intrinsic to Fortran compilers), the Translator was written to provide its own form of recursion and so overcome the normal Fortran failure to naturally support recursion. Recursion was implemented through writing a routine that maintained a call stack and the adoption of elaborate calling conventions used by any subroutine that could be involved in a recursive call sequence. This implementation entailed a substantial amount of bookkeeping, as well as imposing an intermediate call to the stack routine into the call procedure for each recursive subroutine. This procedure was not altered when the Translator was converted from GCOS to Multics. However, Multics has a standard linkage that supports recursion, and the linkage was generated then by the Multics Fortran compiler. Thus the Translator was executing this linkage, but not making use of its recursive abilities. This was corrected by modifying the Translator routines to make direct recursive calls and by deleting the code then used to implement the recursive conventions. Some 45 routines, each making several calls, were involved, and the revision was thus a substantial one. Note that the form

of recursion now used should be readily transferable to other Fortran systems.

After these changes were implemented, another profile of execution time was obtained. This profile revealed that a token analysis routine was taking much more time than seemed justified by its actions. While this routine is very heavily used, the only apparent reason for its large execution time was the overhead time involved in its subsidiary calls to two other simple subroutines. Thus the calls were replaced by the bodies of the subroutines, with a resulting 18% overall improvement in processing time and perhaps a three-fold improvement in the token analysis routine itself. This result argues against the use of small subroutines with Multics, and this idea might well be extended to other parts of the Translator although the gains elsewhere would not be nearly so great as they were in this case.

A limited amount of testing was undertaken in order that the performance of the new Translator could be compared with that of the older version. Such testing is very difficult since each Translator processes a different dialect of the SEMANOL metalanguage, SEMANOL(73) or SEMANOL(76). The metaprograms used for such tests must, therefore, be manually translated from one metalanguage to the other so that the number of errors detected in Translation can be similar in both cases (error treatment affecting processing time). Because of this problem, our comparative timing analysis was not comprehensive. Nevertheless, the results are striking and in agreement with our programmer's subjective evaluation of relative performance. They show a 20-fold improvement in processor time and a 30-fold improvement in elapsed time. Such an improvement is embarrassingly great as it suggests an especially poor initial implementation. However, much of the inefficiency seems to have crept in when conversions were made to Multics, and to new Fortran compilers, without much regard for efficiency. This choice was deliberate, but the difficulty of measuring performance did result in our underestimating the potential efficiency losses that were being incurred.

It is believed that the new Translator is easier to understand, easier to modify, and more machine independent than was the old. Its performance is vastly better, and the translation speed is now about 1,200 SEMANOL(76) lines per minute. This is a much improved product.

Executer Improved - The major change made for better Executer processing efficiency was the implementation of the new SIL format already described. The benefits of the new format were discussed there and largely were observed to result from a more executable code structure that reduced decoding overhead in the Executer. While the control interpretation loop necessarily was totally rewritten, the operator routines ordinarily were changed only to accommodate the changing semantics of the SEMANOL(76) metalanguage. In general, there were few opportunities to rewrite Executer routines in the expectation of realizing substantial execution improvements. Nevertheless, several other changes were made in pursuit of efficiency, and they are briefly described in what follows.

A significant improvement in efficiency was obtained by changing the way in which SIL code is stored internally by the Executer. In the prior implementation, SIL code was intermixed with other data items and referenced indirectly by the use of PL/1 access routines. SIL code is now stored separately in its own array, and so can be referenced directly by Fortran statements. Because the overhead incurred in calling PL/1 routines is high, this new scheme is much faster than the old. This change also has another efficiency benefit in that the garbage collection is now performed more quickly. This benefit results from the fact that the SIL space is no longer a part of the space subject to garbage collection; thus a smaller area is now processed when reclaiming unused space.

The replacement of Fortran input-output routine usage also improved the efficiency of the Executer, although not to the extent that it improved Translator performance. The routines used here were modeled after those implemented earlier for the Translator, so that PL/1 input-output is now used. Note that the drastically revised SIL formats, both external and

internal to the Executer, required that these routines be revised for the new formats. Efficiency was enhanced as part of this process. We should also note that some operator routines were improved. For instance, string concatenation was improved through being changed to expedite a special case occurring more often than originally expected, and initialization was changed to eliminate usage of an external (code equivalence) file. A general improvement was thus realized.

It should be pointed out that providing new operators in SEMANOL(76) had a major influence in improving the efficiency of Executer processing. This was especially true when using the new parse tree operators (e.g., #PARENT-NODE, #ROOT-NODE, #SEQUENCE-OF-ANCESTORS-OF). While equivalent effects could be achieved in SEMANOL(73) by writing more complex expressions, the built-in operators of SEMANOL(76) executed several times more quickly. The built-in operators also made better use of working space and so reduced the need for garbage collection. That is, the older expressions often generated temporary structures as part of their evaluation process, while the new operators do not. The efficiency of expression sought when creating the SEMANOL(76) metalanguage also led to increased metaprogram execution efficiency.

The net efficiency gains achieved in the Executer are difficult to determine, and their source more troublesome yet to obtain. Our limited comparison testing found factors ranging from a 50% improvement to over a 400% factor; a subjective estimate of 50%-100% seems about right. This clearly is a substantial improvement. It is especially significant when one recalls that incremental translation and testing support options are now available to Executer users; a great overall advance in efficiency has thus been provided with the new Executer implementation.

Effective User Interface Implemented

The earlier Interpreter programming system, for SEMANOL(73), provided only rudimentary operational facilities for its use and had no effective metaprogram testing features. While it did offer a form of incremental

translation, incremental translation required several manual steps and so was inconvenient, and time consuming, to use. In order that the new SEMANOL(76) Interpreter programming system might be made operationally more efficient and provide greater convenience for the user, a comprehensive set of user commands was designed and implemented in performance of this contract. The new interface so provided improves user effectiveness, as we have seen in our own metaprogram development.

To some degree, the user interface was shaped by the nature of the host computer system; in this case, by the nature of the Multics system. Thus the user commands for the SEMANOL(76) Interpreter are themselves implemented as Multics-level commands. That is, each user command is a program name that is transformed into a program activation by the Multics operating system. The result of this design is that SEMANOL(76) Interpreter commands are consistent in structure and action with the other commands of Multics. The commands themselves provide the means by which the Interpreter programming system can be used for simple processing, and by which the added features of incremental translation and testing can be controlled. The nature of incremental translation and testing control are explained in what follows.

Incremental translation is essentially the same as what is called "incremental compilation" in other systems. It provides a means by which a metaprogram presently being processed by the Executer can be dynamically modified. Incremental translation is invoked by commands during execution; it causes user-supplied SEMANOL(76) metalanguage statements to be processed by the Translator, as if part of the active metaprogram, and then merged into the metaprogram being processed. Since only changed statements are translated, rather than the entire metaprogram, computer processing time can be reduced during a testing session. As the computational state at the time of incremental translation is saved by the Executer, it is often possible to continue the computation with the new metaprogram without repeating the processing to the incremental translation point. Thus further

time savings are possible. Since SEMANOL(76) metaprograms are often large, since metaprogram interpretation is intrinsically rather slow, and since processing is ordinarily done interactively from remote sites, the savings provided by incremental translation are very helpful.

Implementation of this incremental translation feature was straightforward. The Translator was modified to (optionally) save what is essentially a symbol table when it first processes a SEMANOL(76) metaprogram; it is then able to generate SIL code for isolated statements by reference to this symbol table. The SIL code is thus compatible with the original program context. The SIL code so generated is then read by the Executer and merged into the current metaprogram. This merging is done on #DF or #PROC-DF name, with the consequence that complete semantic definitions are required. While a form of incremental translation had been implemented earlier, this feature was re-implemented as part of the general Translator improvement task; all the Executer additions, plus the writing of the user command programs, were done in the contract performance period.

The testing features represent the addition of an entirely new capability, and all were reflected by modifications to the Executer program. The new test features fall into two classes:

1. Trace features, that are provided so that control flow through a SEMANOL(76) metaprogram can be followed. The trace provides a time-ordered sequence of semantic definition (i.e., function) invocations and returns. This trace can be directed to a file of the user's choice. The user is also given wide control over the content of the trace through several commands that permit naming the specific definitions to be traced and controlling whether subsidiary definitions are traced or not. Trace volume, which can become overwhelming, can thereby be restricted and the test process itself facilitated since the user need investigate only trace information significant to the immediate problem. The current trace status is always available to the user. This is a generous tracing facility.

2. Break features, that are provided so that user interaction with the running metaprogram can be obtained. The essential characteristic of this feature is that the user can establish points within the metaprogram at which processing will be suspended and control relinquished to the user. The user can then interrogate the state of the computation or otherwise interact with the running metaprogram. Break points are set on semantic definitions, and can be easily established or removed by the user through Executer action. Break point status is also readily available to the user. The user action upon a break, apart from altering the break conditions themselves, must be accomplished by the processing of SEMANOL(76) statements. Such statements can be quickly introduced by use of the incremental translation feature, or initially included in the metaprogram to provide this run-time support. This code is then executable upon user command at the break. Thus the SEMANOL(76) metalanguage also serves as the language of user interaction. Associated with this break procedure is an interrupt option that uses the escape mechanism of Multics to provide a user with the ability to suspend Executer processing before the next semantic definition is interpreted. That is, a break is forced at the next convenient point in the metaprogram. A high degree of user control is thereby supplied.

In all, a series of twenty-one user commands was implemented during this performance period. The command set provides the user with a convenient way in which to direct the SEMANOL(76) Interpreter programming system, and makes available an interactive test facility that corresponds closely to those available with the better conventional programming language support systems. The resulting effectiveness of SEMANOL(76) Interpreter users should thereby be increased.

Training Course Presented

A training course, given the title "The SEMANOL(76) System of Programming Language Specification", was presented at RADC in the three day period of 24

May 1977 through 26 May 1977. The course consisted of two phases. Phase I was designed for software managers and was meant to provide a general overview of the SEMANOL(76) system and its possible applications. Phase II was designed for software specialists and was meant to provide a technical foundation for those who might use the SEMANOL(76) system in the future.

The course content and method of presentation were given careful attention, and extensive training materials were prepared by TRW. Eleven documents, containing over 350 pages, were prepared and distributed to attendees. In addition, over 300 vu-graphs were prepared and used in presenting the course. Such comprehensive preparation naturally took much time and effort, but this lengthy preparation enabled TRW to deliver an effective course. The course content is discussed briefly in what follows.

Phase I began with a formal presentation that covered:

1. The Theory of SEMANOL(76). The theory was considered in an informal, generally philosophical, manner rather than being given in rigorous terms. (A formal view is available in published reports.)
2. The SEMANOL(76) Interpreter programming system. This description dealt with the system in terms of its overall processing logic and general utility.
3. The use of SEMANOL(76) in programming language control. The use of a SEMANOL(76) specification as a standard definition was explained in relationship to compiler acquisition, compiler testing, programming language evaluation, change control, programming manual preparation, etc.

The material upon which this presentation was based was made available to attendees in a report called "An Introduction to SEMANOL" by P. T. Berning, copies of the vu-graphs used, and reprints of the Acta Informatica paper "SEMANOL(73): A Metalanguage for Programming Languages" written by E. R. Anderson, F. C. Belz, and E. K. Blum.

Phase I concluded with an on-line demonstration of the SEMANOL(76) programming system running upon the Multics computer. The demonstration was continuous throughout the afternoon, and so allowed course participants ample opportunity for observing system operation and having questions answered.

Phase II immediately followed Phase I, and was conducted on the premise that attendees were compiler writers, and others, who were familiar with formal methods of syntax description and the problems of programming language definition. Phase II was intended to give attendees sufficient understanding of the SEMANOL(76) system so that they would be able to correctly interpret SEMANOL(76) specifications, like that for JOVIAL(J3), and to realize how the SEMANOL(76) Interpreter programming system operates. There was no expectation that they would become competent SEMANOL(76) programmers from receiving this limited amount of instruction; the course was too short for that.

Phase II first introduced the SEMANOL(76) metalanguage. This introduction explained the objects, functions, and relations of SEMANOL(76), and described the different statements provided by the metalanguage. A detailed statement-by-statement explanation was not attempted because of time factors, and because it was felt that attendees could largely gain such knowledge from the SEMANOL(76) Reference Manual by themselves. The explanation was illustrated by use of a trivial formal specification so that some degree of concreteness could be attained. Besides training materials that corresponded to the material presented, a SEMANOL(76) Workbook was also distributed. This Workbook gave many examples of metalanguage use and so provided suggestive illustrations of the nature of the keywords of SEMANOL(76); it was arranged so that the Workbook could be used as a simple self-teaching aid.

Following this introduction to the metalanguage, the SEMANOL(76) method of semantic specification was presented through detailed explanations of a series of specifications of simple programming languages. These languages were based upon BASIC and invented by TRW to serve as tutorial languages.

Complete formal specifications of these three languages were thoroughly examined. In addition, a parse tree representation of a program in the simplest of the tutorial languages was presented, as was the corresponding trace of its interpretation. Thus the operational semantics of one sample program were demonstrated very explicitly. This material was made available to attendees, and should provide a useful basis for independent study of the SEMANOL(76) method. This part of the course gave attendees a feeling for the manner in which we believe formal specifications ought to be written. This information is fundamental to SEMANOL(76) use, but is not covered in the SEMANOL(76) Reference Manual (since it would be inappropriate there); thus this part of the course should have been especially helpful.

Having talked about how SEMANOL(76) could treat simple languages, a brief presentation about the JOVIAL(J3) specification prepared in performance of this contract was given. Examples from the JOVIAL(J3) specification were used to explain how SEMANOL(76) can specify the semantically awkward parts of real-life programming languages. This examination of advanced topics was necessarily somewhat cursory, but did permit time for dealing with specific attendee questions. Handouts of the presented material were distributed.

The final part of the course offered a detailed description of the SEMANOL(76) Interpreter programming system. This description briefly discussed the internal structure of the software, and then concentrated upon giving an explanation of how the programming system might be used. It thus described the various user commands that have become available, with special attention being devoted to explaining the user commands that are helpful when testing SEMANOL(76) metaprograms. A handout was distributed that covered this same material. This session thus imparted some idea of how the Interpreter programming system could be used in practice.

All course presentations were made using vu-graphs, and copies of this material were generally made available to attendees (in addition to other reports). Three TRW instructors participated in making the presentation, and all were available throughout the three days for informal discussion outside established course periods. This condition encouraged casual discussion and so helped insure that questions were answered and course topics well

understood. We should observe that course attendees were themselves highly competent people involved in programming language design and implementation, and so possessed of the background that was being assumed. The net result was a course that we think successfully gave its attendees an accurate picture of the SEMANOL(76) system. It was thus a useful endeavor.

JOVIAL(J3) Specification Written

The JOVIAL(J3) specification written in performance of this project attempts to provide a formal definition of the JOVIAL(J3) programming language described in Air Force Manual No. 100-24 of 15 June 1967. Since AFM 100-24 is incomplete and imprecise, a considerable amount of analysis was performed in order to decide how various unclear features of JOVIAL(J3) were to be treated. The resulting decisions were reflected in the delivered specification, which must thus be recognized as one group's interpretation of AFM 100-24; however, unlike AFM 100-24, it is an interpretation expressed in the formal SEMANOL(76) metalanguage and therefore one that is very exactly stated. The specification thus is a precise description of what we believe JOVIAL(J3) to be.

It should be noted that this analysis was required despite TRW's earlier work on a formal JOVIAL(J3) specification, in contract F30602-72-C-0047, since that earlier work, being more research oriented, had ignored several complicating aspects of JOVIAL(J3). In particular, compools and overlay were disregarded in the first formal specification. Because of this, and because of changes to the SEMANOL metalanguage and changes to TRW's semantic description style, the earlier specification was of little use in this project. The JOVIAL(J3) specification written in this project managed to retain portions of the earlier context-free grammar, but is otherwise an entirely new product.

The JOVIAL(J3) specification that was prepared is syntactically complete, as the context-free grammar includes all the features of JOVIAL(J3), but is semantically incomplete in that the execution effects of presets, the ALL loop termination clause, and file input-output are not fully prescribed. In addition, the context-sensitive constraints included in the specification do not include all those conditions that are suggested in AFM 100-24. In part, this stems from the fact that AFM 100-24 is often ambiguous with regard to whether certain erroneous conditions are to be detected if they do not occur during execution; that is, whether they are to be determined at compile time or in execution. None of these missing features present special semantic

description problems; their omission simply reflects our inability to prepare a complete specification within the performance period.

The SEMANOL(76) metalanguage in which the JOVIAL(J3) specification is written provides a formalized notation to be used for describing the syntax and semantics of programming languages. The metalanguage, and certainly the underlying theory of semantics and the specification conventions used with SEMANOL(76), result in a programming language being described in operational, interpretive, terms. Indeed, it is fair to think of the SEMANOL(76) metalanguage as a programming language meant to be used in writing programs that are interpreters of source text strings of the language being defined. Since the JOVIAL(J3) specification is such a program, it is described here largely in its role as a computer program that processes JOVIAL(J3) program text.

As with any conventional document or computer program, the manner in which the JOVIAL(J3) specification was written reflects a certain individualism, even though constrained by conventions and the application of standardized techniques. Hence, the JOVIAL(J3) "specification program" presented here is certainly not the only one that could be allowed; nor is it likely to satisfy all readers with regard to its readability. But it is the product of very thorough analysis and thoughtful design, and so reflects our understanding of the JOVIAL(J3) programming language that is intended by AFM 100-24.

The SEMANOL(76) description of JOVIAL(J3) is written as a prescription of a sequence of processing steps that one can follow, for a potential JOVIAL(J3) system (i.e., program, compools, and library) and input to the system, in order to determine:

1. Whether the given system is a legal JOVIAL(J3) system.
2. The effect of executing a legal JOVIAL(J3) system upon its input.

Since the description is a metaprogram, the description assumes a certain structure as shown in Figure 3. The left side of that diagram shows the transformations which are made to the representation of the JOVIAL(J3) system text as interpretation is performed. The central block diagram is a simple flowchart showing the series of processing steps that the SEMANOL(76) meta-

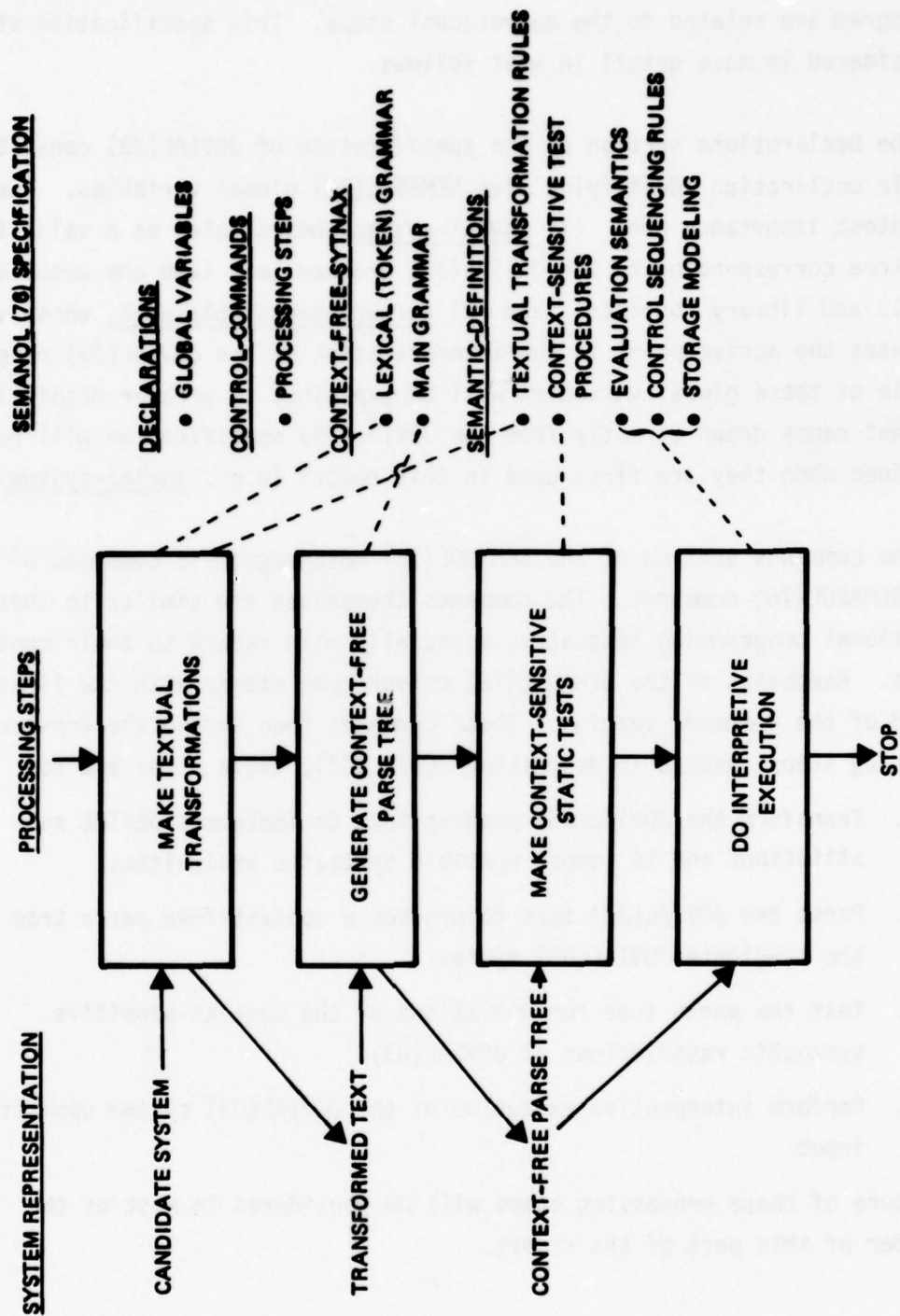


FIGURE 3: JOVIAL (J3) SPECIFICATION STRUCTURE

program describes. The right side of the diagram reflects the static structure of the SEMANOL(76) metaprogram and the way in which the various parts of the metaprogram are related to the operational steps. This specification structure is considered in more detail in what follows.

The Declarations section of the specification of JOVIAL(J3) consists of a single declaration identifying five SEMANOL(76) global variables. The two of greatest importance are: (1) jovial-system, which takes as a value the parse tree corresponding to the JOVIAL(J3) program text (and any associated compools and library routines), and (2) current-executable-unit, whose value designates the active point in the interpretation of the JOVIAL(J3) program. The role of these global variables will be explained in greater detail later. Note that names drawn directly from the JOVIAL(J3) specification will be underlined when they are first used in this report (e.g., jovial-system).

The Commands section of the SEMANOL(76) metaprogram is composed of only a few SEMANOL(76) commands. The commands themselves are similar to those of conventional programming languages, especially with regard to their control aspects. Execution of the SEMANOL(76) metaprogram starts with the first command of the Commands section. These commands then define the top-level processing steps invoked in describing JOVIAL(J3); these steps are to:

1. Transform the JOVIAL(J3) program text to implement DEFINE substitutions and to remove possible syntactic ambiguities.
2. Parse the JOVIAL(J3) text to produce a context-free parse tree of the candidate JOVIAL(J3) system.
3. Test the parse tree for violations of the context-sensitive syntactic restrictions of JOVIAL(J3).
4. Perform interpretive execution of the JOVIAL(J3) system upon its input.

The nature of these processing steps will be considered in most of the remainder of this part of the report.

The textual transformation step is required in order to eliminate certain context-free ambiguities and to describe the effect of DEFINE directives and their invocations. The context-free ambiguities are legal constructions of JOVIAL(J3) whose existence interferes with the construction of an unambiguous, semantically useful, grammar for the JOVIAL (J3) system. Notable examples of this problem are hollerith and transmission-code constants. For example, natural grammars for JOVIAL(J3) would allow the statement

AA = 10H(X)\$BB=1H(X)\$

to be parsed as either a single assignment statement or as two assignment statements, since the value of the count field of the constant cannot be used in context-free grammars to resolve the ambiguity. This problem is solved in the SEMANOL(76) metaprogram by transforming the bracketing parentheses of each such constant into unique delimiters not allowed within the constant text itself. This transformation is defined in terms of the string operations of SEMANOL(76). These troublesome constructions are thus changed into forms that can be naturally defined in a context-free grammar.

DEFINE directives and their invocations are a form of macro definition and macro call, which again are difficult to incorporate into a semantically useful context-free grammar for JOVIAL(J3). Therefore, another transformation step is included in the metaprogram which effects all DEFINE substitutions. The text is first parsed with respect to a lexical (token) grammar that appears as the Lexical Syntax in the context-free syntax section of the specification. The resulting parse tree's principal structures are nodes representing tokens and the gaps (of blanks) between tokens. The SEMANOL(76) sequence construction operators are then used to form a sequence of the token and gap nodes in the tree. The nodes of this sequence are scanned, left-to-right, and a new sequence of token and gap nodes is formed in which each token corresponding to a DEFINE invocation is replaced by its expansion. The second sequence is then converted back to a string, which is the original text as revised by the DEFINE substitutions. In total, these two sets of transformations produce a string that is consistent with a natural context-free grammar; they are defined in the Lexical Analysis part of the specification.

Following the textual alterations, the transformed program text is parsed. This parse process is invoked by the SEMANOL(76) operator #CONTEXT-FREE-PARSE-TREE and is directed by the jovial-j3-system grammar given in the Main Syntax section of the specification. The product of this operation is a parse tree representation of the JOVIAL(J3) system, or possibly an error condition if the grammar can lead to more than one parse of the given text (i.e., the grammar is ambiguous) or if the text cannot be parsed. The parsed representation reveals the structure of the system and so is a convenient basis upon which to formulate the later semantic description. The lexically transformed text itself is retained as the terminal leaves of the parse tree.

The SEMANOL(76) syntactic definitions used to define the grammar look much like the productions of the usual treatments of such programming language grammars. For instance, the following is a part of the main grammar of JOVIAL(J3):

```
#DF program => <start-statement><gap><statement-list>
               <gap><term-statement>#.

#DF statement-list => <statement-list-element>
                    <%<<gap><statement-list-element>>>#.

#DF statement-list-element => <statement>#U<declaration>#U<directive>#.

#DF gap => <#GAP>
          => <#GAP><special-separators><#GAP>#.
```

This grammar defines a program as beginning with a start-statement, ending with a term-statement, and containing at least one statement, declaration, or directive. The #GAP keyword designates strings of zero or more blanks, with the condition that one blank is required between alphanumerics to the left and right. The % designates zero or more occurrences of gap, statement-list-element pairs. Note that gap may include separators, which are later defined to include comments, as well as blanks.

The parse tree representation produced by this step is assigned to the SEMANOL(76) global variable jovial-system. Since no other assignment is made to this variable, subsequent processing steps may use the SEMANOL(76) name jovial-system as a constant identifier for the tree.

The next processing step is the imposition of syntactic restrictions that cannot be expressed in a context-free grammar. That is, not all systems that can be parsed using the context-free grammar are legal JOVIAL(J3) systems. It is the intent of this section of the specification to provide an operational algorithm to detect such illegal programs before an attempt is made to interpret them. Since the application of these tests is made before interpretation proper, they can cause the rejection of programs that could be interpreted without encountering the error condition. That is, these tests correspond to those that a compiler might make and the consequences can be semantically different than if similar tests were applied at execution time.

In the JOVIAL(J3) specification, tests are made to insure that go-to designators are names appearing on statements, programs, closes, or switches; that RETURN appears only in procedure declarations; that loop variables are distinct; that ODD modifiers are not applied to floating variables; etc. These tests make use of the sequence operators of SEMANOL(76) and the #THERE-EXISTS iterator in their formulation; they appear under the title Context Sensitive Checks in the specification.

The process of interpretively executing the JOVIAL(J3) system is itself subdivided into the distinct elements shown in Figure 4. The first step in this process to establish the sequence-of-executable-units-in the JOVIAL(J3) program. Each executable unit is defined by the context-free grammar so as to be a node in the parse tree for the program, and each is associated with a specific execution event. In JOVIAL(J3), the executable units are chosen at a very fine level of detail since nodes for every variable reference and operation are included, as well as nodes for pure control events, such as jumps. This fine level is needed so that abnormal returns, which can interrupt expression evaluation, can be described conveniently (i.e., it is a way to specify a semantically awkward element of JOVIAL(J3)).

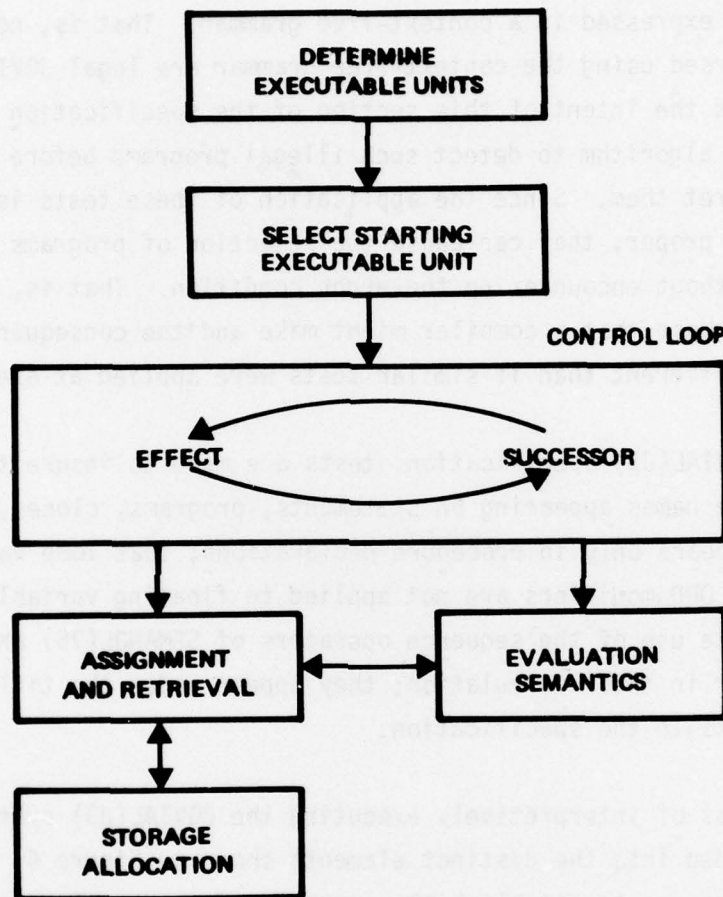


FIGURE 4. INTERPRETIVE EXECUTION STRUCTURE

Although the level of detail is very low, the construction of the sequence-of-executable-units-in is specified in a hierarchical manner. First, the executable statements are identified, then the portions of these statements which have executable units, and finally the executable units themselves. Note that, for JOVIAL(J3), this means execution semantics tend to be specified at something less than the statement level, but that the hierarchical decomposition used here makes the relationships clear.

The next step of interpretive execution is locating the unit at which JOVIAL(J3) program execution is to start; this is the first executable unit in the statement designated in the term-statement of the JOVIAL(J3) program, if one is so designated, or is otherwise the first unit of the sequence-of-executable-units-in the program. This node is then assigned to the SEMANOL(76) global variable current-executable-unit. At any future point in the interpretation of the JOVIAL(J3) system, this global variable serves to identify the active point in the JOVIAL(J3) system, and transitions of control are specified as changes in this value. Thus the variable current-executable-unit serves a function analogous to that of the instruction counter of a conventional computer except that, in this case, control is defined in terms of the program text, rather than in terms of a machine storage structure. Once the starting executable unit is assigned to current-executable-unit, the preliminary steps are completed and execution proper may begin.

The body of interpretive execution is accomplished within a high level control loop. In this loop, (1) the effect of executing the current-executable-unit is described and (2) a successor to the current-executable-unit is determined and made the new current-executable-unit. The loop is terminated, for example, when the current-executable-unit is a node whose associated effect is to halt the computation, as with the STOP statement of JOVIAL(J3). In this case, the control loop is ended normally with #COMPUTE! #STOP.

Supporting this control loop are evaluation semantics that apply to executable units, called evaluation units, corresponding to the operations and primitive operands of JOVIAL(J3) expressions. For instance, the parse tree for the subexpression "AA+BB" contains three evaluation units: one <sum> node, node i, and two <simple-variable> nodes, node j and node k, representing "AA" and "BB" respectively (See Figure 5). These nodes appear in the sequence-of-executable-units-in postorder, or operator postfix order; thus the operand nodes for "AA" and "BB" are followed by the operator <sum> node. The semantic selector definitions, such as operand1-of and operand2-of, define the relationships among these nodes; so node j is operand1-of node i and node k is operand2-of node i in the sequence-of-executable-units-in. Associated with each evaluation unit is a unique SEMANOL(76) global variable which receives the result of evaluating that unit. These evaluation unit variables permit ready interruption of expression evaluation as may be required when modeling abnormal returns.

Evaluation of the <sum>, node i, then proceeds as follows: the latest value of the unique variable associated with operand1-of node i is added to the latest value associated with operand2-of node i, and the result value is assigned to the unique variable associated with node i. The precise meaning of this addition is determined by the type of the <sum> node, which in turn depends upon the types of its operands. If the <sum> node is of fixed or integer type, then the meaning of addition also depends upon the attributes associated with each relevant evaluation unit. The derivations of type and attributes are thus defined for each class of evaluation unit. The attributes are represented by a sequence of three integers for the numbers of integer bits, fractional bits, and minimal bits that apply to the value of the evaluation unit.

The values for the numeric evaluation units are kept in implementation numeric form, and the operations, such as addition, that produce numeric result values from numeric operand values, are implementation defined operations which take implementation numeric form values as arguments. In the case of fixed point operations, the attributes of both operands and the

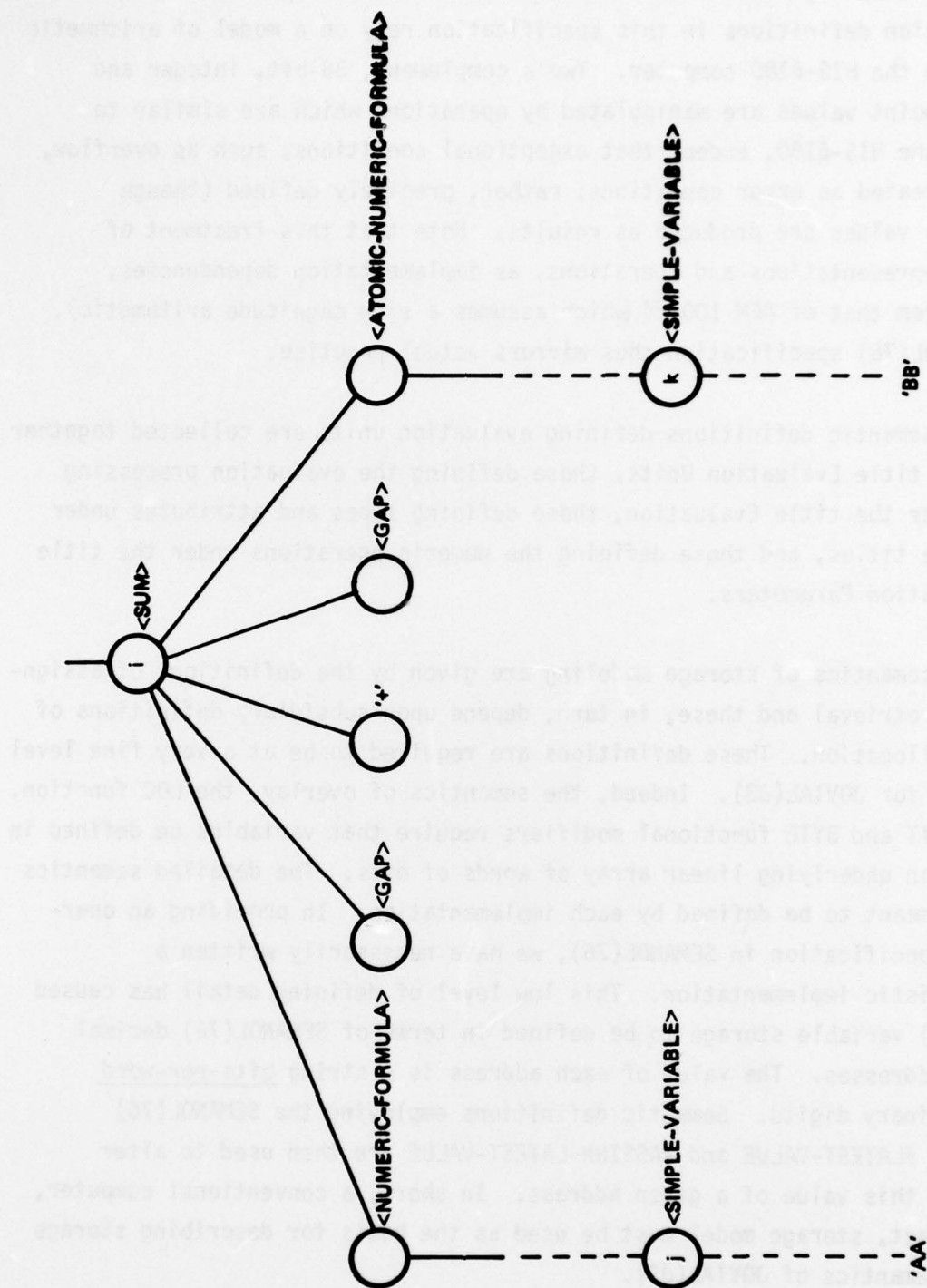


FIGURE 5. EXPRESSION REPRESENTATION

result are also arguments of the implementation defined operation definitions. The operation definitions in this specification rely on a model of arithmetic based upon the HIS-6180 computer. Two's complement, 36-bit, integer and floating point values are manipulated by operations which are similar to those of the HIS-6180, except that exceptional conditions, such as overflow, are not treated as error conditions; rather, precisely defined (though arbitrary) values are produced as results. Note that this treatment of numeric representations and operations, as implementation dependencies, differs from that of AFM 100-24 (which assumes a sign magnitude arithmetic). The SEMANOL(76) specification thus mirrors actual practice.

The semantic definitions defining evaluation units are collected together under the title Evaluation Units, those defining the evaluation processing steps under the title Evaluation, those defining types and attributes under those same titles, and those defining the numeric operations under the title Implementation Parameters.

The semantics of storage modeling are given by the definitions of assignment and retrieval and these, in turn, depend upon subsidiary definitions of storage allocation. These definitions are required to be at a very fine level of detail for JOVIAL(J3). Indeed, the semantics of overlay, the LOC function, and the BIT and BYTE functional modifiers require that variables be defined in terms of an underlying linear array of words of bits. The detailed semantics are thus meant to be defined by each implementation. In providing an operational specification in SEMANOL(76), we have necessarily written a characteristic implementation. This low level of defining detail has caused JOVIAL(J3) variable storage to be defined in terms of SEMANOL(76) decimal integer addresses. The value of each address is a string bits-per-word long of binary digits. Semantic definitions employing the SEMANOL(76) key words #LATEST-VALUE and #ASSIGN-LATEST-VALUE are then used to alter or recall this value of a given address. In short, a conventional computer, non-abstract, storage model must be used as the basis for describing storage related semantics of JOVIAL(J3).

At the highest level of abstraction, each variable reference in the JOVIAL(J3) program is associated with a standard-reference-address. Every JOVIAL(J3) assignment is then accomplished by a generalized-assign-latest-value function, which takes a value and a standard-reference-address as arguments. Similarly, generalized-latest-value, when applied to a standard-reference-address, yields a value from JOVIAL(J3) storage. Both of these operations can be thought of as abstract operations in the absence of an interest in any explicit or implicit overlaying affecting the referenced variable. However, in the case of overlaying, it is necessary to consider the detailed definitions of these operations (which appear under the title Generalized Assign). Both operations apply the field-descriptor-sequence-implied-in operator to the standard-reference-address to obtain a sequence of field descriptors; each field descriptor is itself a sequence of three integers identifying a word-address, a bit location, and a field length. Taken together, the series of fields so described defines the total storage area allotted to the variable, and does so in a way that permits the non-contiguous storage allocation that is needed for JOVIAL(J3). Each field descriptor is then given to low level definitions for alteration or extraction of a particular field in a particular word. Thus, eventually, the semantics of JOVIAL(J3) variables are given in terms of a sequence of fields within words.

As observed, any definition of a standard-reference-address is inherently an implementation defined feature of JOVIAL(J3). In the implementation modeled by this specification, library routines are modeled, as well as compools with common blocks. In the SEMANOL(76) specification, nodes representing the library subprograms, the common blocks, and the JOVIAL(J3) program are used to designate these independent addressing units. These are collected in a sequence in a given order, which can easily be permuted to correspond to an alternate implementation, and allocated as blocks in JOVIAL(J3) memory so that the spaces do not overlap.

Within addressing units, variable addresses are formed in terms of an address relative to the start of the addressing unit of which the variable is a part. (Note that only the word-address part of these addresses actually

varies in the transformations to be described.) Within an addressing unit, variable references may be made to variables for which space is to be allocated or not. Variables for which space is to be allocated include non-overlapping simple items, tables, and loop variables. Variables for which space is not to be allocated include table items and overlapping simple items and tables; in these cases, related variable references will cause space to be reserved. Thus if a name satisfies is-prime-overlay-initiator (i.e., the name is the first variable name mentioned in a collection of related overlay declarations), it reserves space for the entire overlay block. In the SEMANOL(76) specification, nodes representing the declaring occurrence of the names of variables for which space will be allocated are collected in a sequence, and are then given relative addresses (in order) in a manner which guarantees that their spaces do not overlap.

Then every allocated variable reference takes its relative address to be that of its declaring occurrence. For every overlapping simple item or table reference, the process of determining relative addresses is carried a step further so that relative addressing within the block is done. That is, addresses are computed relative to the start of the enclosing overlay block. Similarly, table items are given addresses relative to the entry of which they are a part. Then for a reference to a table item, the entry relative address is modified by the index-offset, computed from the index in the reference, and further modified by the standard-reference-address of the table containing the reference table item. Relative addresses are thereby made absolute so that JOVIAL(J3) storage may be accessed.

Note that a relative address or a standard-reference-address have a similar representation. This representation is as a sequence of three elements: a field-descriptor for the first field of the variable (as described earlier), a next field descriptor, and a count of the number-of-bits in the variable. The next field descriptor provides the information needed to step through the fields of the variable, while the bit count simply defines the variable extent. This information can be used to construct the field-descriptor sequence, and otherwise used to derive information about where a

variable is located in JOVIAL(J3) storage. The descriptions of address derivation are found in the specification sections called Standard Ref Addr, Relative Addresses, Addressing Units, and Addr Unit Addresses.

In all references to variables, it is important to establish the declaration-for the variable actually being referenced. This, of course, depends upon the scope rules for JOVIAL(J3), and these are included in the SEMANOL(76) specification by an implicit sequence of scoping contexts that control the search for the desired declaration. For instance, in order to find the declaration-for a variable name in a procedure declaration of a JOVIAL(J3) program: first the body of the procedure declaration is searched, then the body of the JOVIAL(J3) program, then the compools referenced in the control input of the program and, finally, a defaults context in which the system default mode-directive and system default REM and REMQUO definitions are given. The semantic definitions for this declaration determination are given under the titles Scoping Contexts and Names.

Some elements of control semantics have already been discussed in describing the methods by which the executable units are distinguished and by which the executable unit at which JOVIAL(J3) program execution starts is selected. More control details are then given in successor functions for the various classes of executable units of JOVIAL(J3); these appear in the Control section of the specification. Control in JOVIAL(J3) poses a difficult semantic description problem because programmer defined functions can employ "abnormal returns"; i.e., precipitous termination of their computation followed by an abrupt change in the flow of control to a JOVIAL(J3) statement other than the one containing the aborted function call. The problem is that evaluation, which is ordinarily considered a process which once begun would always complete, has become something that can be interrupted at almost any point.

There are two basic possible solutions to the problem: (1) provide a metalanguage exception handling facility to be employed in this circumstance or (2) write the specification so that every evaluation step is guaranteed to complete once it has begun. Prior versions of the SEMANOL metalanguage did

have such an exception handling facility in the #RESUME(;n) primitive; however, as noted previously, this feature was removed from SEMANOL(76) because of the difficulty of clearly describing it. Without this exception handling primitive, it becomes necessary to insure completion of every evaluation step. This could have been done by creating special exception flags, and providing tests of the flags in each subsidiary definition to allow partial evaluation of the metalanguage definitions. This option appeared cumbersome in practice and so was rejected. The accepted approach was thus to have non-interruptable executable units.

In the SEMANOL(76) specification of JOVIAL(J3), the mesh of executable units is very fine in that each operator and primitive operand is treated as an executable unit. The intermediate results of every operation are then held in special variables associated with these evaluation units. With this approach, an abnormal return can be described as easily as can other control transfers of JOVIAL(J3).

The entire delivered JOVIAL(J3) specification was processed without error message by the SEMANOL(76) Translator program and, therefore; each command, semantic definition, and syntax definition is syntactically correct; all referenced procedures, definitions, and variables are defined; no name is multiply defined; etc. In effect, the specification contains no errors of the kind that compilers ordinarily can detect. However, interpretive execution testing, using JOVIAL(J3) programs, could not be comprehensively conducted because of a lack of time. The context-free grammar was carefully tested. The semantics of control were tested thoroughly, using prototype skeleton versions of evaluation and assignment, as were integer and fixed point evaluation semantics. The lexical transformations were also well tested. Storage modeling (i.e., assignment and allocation) was not tested in its final version, and this was the major defect in the testing process that was conducted. These facts should be remembered when reading the delivered specification.

It should be noted that SEMANOL(76) is a rarity in that it is a semantic description method that permits specifications to be thoroughly machine tested. At best, most other formal methods seem to allow only compiler-like testing to be done (although this feature is not ordinarily implemented for systems other than SEMANOL(76)). Thus, while testing was less complete than we would have liked, it still has produced a much sounder product than would be likely with a non-operational method.

CONCLUSIONS AND RECOMMENDATIONS

The improvements made to the SEMANOL system in performance of this contract have produced a system that is suitable for practical applications in programming language development and control. This has been achieved by:

1. Improving the metalanguage, to create SEMANOL(76), so that the economy and clarity of expression possible with its use can be enhanced. A more efficient notation, for both people and computer processing, has thereby been realized.
2. Improving the processing efficiency of the Interpreter program so that its speed now seems adequate for developing and testing specification metaprograms.
3. Improving the user interface with the operational system, so that the Interpreter is now convenient to use, offers a wide range of user commands, and includes a comprehensive interactive metaprogram testing facility.

The usefulness of the system is thus substantially better than it had been.

The training course, while helpful to those who attended it, has a more enduring value as well since the materials developed for that course can serve as the basis for the more extensive training program that must accompany any effort to use SEMANOL(76) more widely. The SEMANOL(76) specification of JOVIAL(J3) demonstrates the capability of SEMANOL(76) to fully describe a typical programming language; it also provides a basis from which a formal accepted standard for JOVIAL(J3) could be developed.

Further improvements to the SEMANOL(76) system remain possible and ought to be considered. Processing efficiency can certainly be bettered still more by essentially moving in the direction of an implementation using compilation instead of interpretation. In such a implementation, the Translator would

generate "executable" code (perhaps in PL/1), while the interpretive control functions of the Executer would be removed (although much of the run-time support of the Executer would be retained). The new SIL format is already a step in this direction, as were other changes made to the Translator. An incremental approach is envisioned, since existing Executer routines can be replaced by Translator generated code on a continuing basis as the expected execution benefits of replacing any given routine are found to justify the replacement.

The preparation of a more extensive training course and tutorial materials is also an area in which further work is appropriate; the availability of the current materials will aid in this effort. Future development of formal specifications should be done for any language over which active control is to be exercised; unfortunately, it does not seem that JOVIAL(J3) is a likely candidate programming language for such control. The SEMANOL(76) metalanguage should be subjected to continuing evaluation, especially by the new users, even though the #RESUME(;n) feature is the only major element of the metalanguage that we feel warrants reconsideration.

While the current SEMANOL(76) system is a useful one, further development can make it more attractive yet and insure its suitability for Air Force applications of formal specification methods.

MISSION **of** **Rome Air Development Center**

RADC plans and conducts research, exploratory and advanced development programs in command, control, and communications (C³) activities, and in the C³ areas of information sciences and intelligence. The principal technical mission areas are communications, electromagnetic guidance and control, surveillance of ground and aerospace objects, intelligence data collection and handling, information system technology, ionospheric propagation, solid state sciences, microwave physics and electronic reliability, maintainability and compatibility.

